9-2013

DriverGuard: Virtualization based fine-grained protection on I/O flows

Yueqiang CHENG
Singapore Management University, yqcheng.2008@smu.edu.sg

Xuhua DING
Singapore Management University, xhding@smu.edu.sg

Robert H. DENG
Singapore Management University, robertdeng@smu.edu.sg

DOI: https://doi.org/10.1145/2505123

DriverGuard: Virtualization-Based Fine-Grained Protection on I/O Flows

YUEQIANG CHENG, XUHUA DING, and ROBERT H. DENG,
Singapore Management University

Most commodity peripheral devices and their drivers are geared to achieve high performance with security functions being opted out. The absence of strong security measures invites attacks on the I/O data and consequently posts threats to those services feeding on them, such as fingerprint-based biometric authentication. In this article, we present a generic solution called DriverGuard, which dynamically protects the secrecy of I/O flows such that the I/O data are not exposed to the malicious kernel. Our design leverages a composite of cryptographic and virtualization techniques to achieve fine-grained protection without using any extra devices and modifications on user applications. We implement the DriverGuard prototype on Xen by adding around 1.7K SLOC. DriverGuard is lightweight as it only needs to protect around 2% of the driver code's execution. We measure the performance and evaluate the security of DriverGuard with three input devices (keyboard, fingerprint reader and camera) and three output devices (printer, graphic card, and sound card). The experiment results show that DriverGuard induces negligible overhead to the applications.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: Virtualization, hypervisor, I/O data protection, untrusted OS, trusted path

ACM Reference Format:
Cheng, Y., Ding, X., and Deng, R. H. 2013. DriverGuard: Virtualization-based fine-grained protection on I/O flows. *ACM Trans. Inf. Syst. Secur.* 16, 2, Article 6 (September 2013), 30 pages.
DOI:http://dx.doi.org/10.1145/2505123

1. INTRODUCTION

Device drivers are often blamed as the main cause for system failures and security breaches, mainly due to their enormous code size and the much higher bug rate than other kernel code [Chou et al. 2001]. Various schemes have been proposed to improve system reliability by isolating driver errors (e.g., Nook [Swift et al. 2003] and SafeDrive [Zhou et al. 2006]), or to defend against device I/O misuses for illegal memory accesses (e.g., BitVisor [Shinagawa et al. 2009] and the schemes in [Willmann et al. 2008]). In this article, we study the other side of the coin: how to protect the I/O data, which is motivated by attacks on sensitive I/O data, such as password keystrokes, fingerprint templates, sensor readings and confidential printouts.

This work was funded in the Singapore Management University through the research grant MSS11C004.
An early version of this article was presented in part at ESORICS 2011 [Cheng et al. 2011].
Authors' address: Y. Cheng, X. Ding, and R. H. Deng, School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902; email: yqcheng.2008@phdis.smu.edu.sg, {xhding, robertdeng}@smu.edu.sg.

DOI:http://dx.doi.org/10.1145/2505123

As compared to applications and other kernel components such as system call functions, driver operations or I/O flows are more attractive to malwares targeted at sensitive data for the following reasons. First, there exist more loopholes to exploit due to the complexity of I/O mechanisms and the abundance of driver bugs. For instance, IRQ number sharing allows a malicious interrupt handler to easily access another handler's data. Second, most drivers handle raw data generated by or for hardware. In many applications, raw data are more favorable to attackers as compared to derived data. For instance, a user's fingerprint template is lifelong valid whereas a secret key derived from the fingerprint template may remain valid only for a few hours. Furthermore, most commodity I/O devices nowadays are not encryption capable and raw data are exposed to any code accessing them.

We aim to protect data flows between applications and devices against an untrusted kernel throughout the entire I/O lifecycle. In particular, we focus on those devices that render raw data, that is, sound cards and printers, or generate raw data for applications, that is, seismic sensors and fingerprint scanners. We are less concerned with disks and network adaptors, because these devices deal with derived data from applications. Therefore, a straightforward solution to protect the disk I/O and the network I/O is to encrypt the data before and after I/O operations.

In this work, we present DriverGuard, a holistic and compact I/O protection system making use of a combination of cryptographic and virtualization techniques. We implement DriverGuard with slight changes on the device drivers and the Xen hypervisor. Our experiments with several I/O devices demonstrate that DriverGuard imposes little overhead to the system and causes unnoticeable delays to user applications. DriverGuard is complementary to many user space protection schemes such as Overshadow [Chen et al. 2008], and SP^3 [Yang and Shin 2008]. A composition of DriverGuard and a user-space protection scheme can protect the whole lifecycle of data processing.

Our work is also remarkably different from secure I/O [Shinagawa et al. 2009] and driver code security [Seshadri et al. 2007]. Secure I/O copes with those attacks misusing the I/O mechanism (especially DMA operations) for illegal memory accesses. Driver code security tackles software attacks, such as return-address attacks [Checkoway et al. 2010] and code injection attacks [Lineberry 2009], which gain the root privilege by subverting drivers. Although these attacks do not necessarily target at the I/O data, they are one of the threats considered in our study. Our work is similar to the trusted path proposed by Zhou et al. [2012]. Their trusted path aims to assure the *secrecy* and *authenticity* of data transfers through a trusted path from the *new inserted user-level* driver to the device, while our work focuses on the protection of the *secrecy* of the I/O data through a trusted path built upon the *legacy* drivers.

Organization. The next section describes some background knowledge. Then, we define the problem as well as the threat model, security requirements and main challenges in Section 3, and explain the design rationale in Section 4. We describe the design overview, privileged code block and design details of DriverGuard in Section 5, 6 and 7, respectively. Section 8 discusses the automatical PCB identification and the full path I/O protection, and Section 9 shows the evaluation of DriverGuard through experiments and performance measurements. Finally, we give the related work in Section 10 and conclude the paper in Section 11.

2. BACKGROUND

In this section, we introduce the related background knowledge (i.e., device configuration and I/O mechanisms) on x86 platforms to facilitate the understanding of our design.

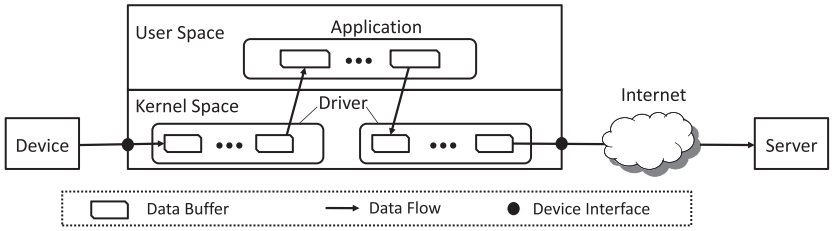


Fig. 1. A typical I/O flow.

2.1. Device Configuration Register Access

The physical addresses and the I/O ports of all devices are decided by the device configuration registers. All these configuration registers are located in the northbridge chipset [Fleming 2008]. There are two possible methods to access them. One is through I/O ports. The I/O port *CONFIG_ADDRESS* (i.e., 0x*CF*8) is used to select a dedicated device whose configuration space is updated through the I/O port *CONFIG_DATA* (i.e., 0x*CFC*). The other method is through MMIO. Typically there is a continuous 256-MB memory region reserved by the system for all devices. Any access to this region will trigger the chipset to propagate the configuration throughout the whole system. In order to avoid the configuration space conflicts (e.g., MMIO mapping attack) between different devices, the privileged code (e.g., the hypervisor) is able to verify the update requests by setting access control on the above I/O ports and the reserved memory mapped region.

2.2. I/O Mechanism

The I/O subsystem is a well-known thorny component of the kernel, due to the complexity and the heterogeneity of hardware and drivers. We only offer a high-level view on the *typical* I/O mechanism in a Linux platform without drilling down to the details.

When a system boots up, the kernel scans all attached devices and creates an array of device structures, each of which contains the physical address of a device and a *driver_pointer* among other important information. If the device supports port-mapped I/O, the CPU can issue commands to the device through its I/O ports, which is in an address space separated from the linear address space. If the device supports memory-mapped I/O, its physical registers are mapped into one or more reserved physical regions. A loaded driver first registers to a device by setting the *driver_pointer* of the corresponding device structure to itself. The driver explicitly requests the kernel to allocate I/O ports or memory-mapped regions in the virtual space. The driver installs a handler on an interrupt identified by an IRQ number, which can be shared with other device handlers.[1] In that case, there exists a handler queue for this interrupt. This completes the initialization phase and the drivers are ready to offer I/O services.

In the following, we use a user (Alice) login procedure to illustrate a typical I/O process (see Figure 1). When Alice attempts to login to a web service with her private credentials, such as password, face picture or fingerprint data, the browser issues a system call to retrieve the raw data inputs from the corresponding device. There is a device driver responding to the system call. The device driver allocates one or more data buffers[2] and choose one of them to get the raw data from the device. The data transferring between a device and the system is through the device interface. The

[1]For certain buses, such as PCI, it is mandatory to share the IRQ number among devices connecting to the bus.

[2]For keyboard, usually there is only one data buffer. For camera, normally there are several data buffers.

device interface can be MMIO or PIO or DMA. For instance, the keyboard driver handles the data transferring via DMA for a USB-keyboard and PIO for a PS/2-keyboard, respectively. After receiving the raw data from the device, the device driver may translate the data into another buffer with a predefined format or just simply move the data from one buffer to another, until the data is forwarded to the browser data buffer. Getting needed data, the browser may do certain computations, and sends the derived data to the remote server via the network channel.

3. PROBLEM DEFINITION

In this section, we state our goals together with the threat model, and present the possible attacks and the main challenges we are facing.

3.1. Our Goals

There are several approaches [Chen et al. 2008; Yang and Shin 2008] proposed to protect the data in the user space. Therefore, in this article, we focus on the I/O flow protection in the kernel space. Specifically, we attempt to propose an approach to defend against attacks that get the value of the I/O data from device interfaces and kernel-space buffers. More specifically, we focus on the protection of the *raw I/O data* that is generated from or send to devices. Disk I/O and network I/O are not in the scope of our study, because neither disks nor network adaptors produce or render raw data. In fact, data stored in disks or transmitted across networks are actually generated by user applications. Therefore, they can be protected using user-level encryption techniques (e.g., using SSL to protect the network data). Note that the devices mentioned in this article are hardware/physical devices rather than virtual devices.

The second goal is to propose a *generic solution* to protect all kinds of I/O flows on different machines and different devices. We attempt to make our system to be compatible with commodity operating systems and legacy applications.

3.2. Threat Model and Assumptions

In our threat model, we consider the malicious software residing in the guest as the adversary. The malware may compromise the kernel through attacks like ROP [Buchanan et al. 2008; Checkoway et al. 2010; Shacham 2007] and code injection [Lineberry 2009]. As a result, the adversary can take full control of the guest, for example, launching arbitrary code and issuing any DMA requests.

We assume that malware can not subvert the hypervisor. This assumption is reasonable since the TPM-based secure boot scheme can guarantee the load time integrity of the hypervisor, and the virtualization technology can prevent malicious software and illicit DMA accesses driven by the guest OSes in runtime. In addition, some proposed hypervisor-protection schemes [Azab et al. 2010; Rafal et al. 2008; Wang and Jiang 2010; Wang et al. 2010] can be applied to ensure the hypervisor's security.

We trust the end user, and assume that the adversary can not physically control the system. We assume that the hardware devices always behave according to their specifications. We also assume the system firmware is trusted as in Intel [2008], Kun et al. [2012], Phoenix Technologies [2006], and Vasudevan et al. [2012]. The modern BIOS has a built-in hardware lock mechanism to set itself as read-only and only accepts signed updates, so that the OS cannot tamper with it. Due to the complexity of the x86 platform (e.g., optional ROM), this assumption may not always be true. Nonetheless, it is still possible to validate the system firmware by the proposed attestation approach [Li et al. 2011] or by a trusted system integrator. Furthermore, for the computers in an organization, the security-savvy system administrator can simplify the system boot settings, such as disabling unnecessary option ROMs.

In this work, we only focus on uniprocessor platforms. Attacks from multi-core or multi-processor are out of scope of our discussion. Note that it is not our interest in this paper to study how to check the trustworthiness of a device driver. We assume that a trusted authority[3] signs every device driver to be installed. The hypervisor can validate the integrity of device drivers by verifying the signatures. Neither the denial-of-service attacks nor the side channel attacks (e.g., Song et al. [2001]) are in the scope of our work.

Although the security and functionality of DriverGuard are independent of user space protection, the benefits are maximized if DriverGuard joins schemes such as Overshadow [Chen et al. 2008] and SP[3] [Yang and Shin 2008] to safeguard the entire I/O data life cycle covering both kernel and user spaces. We will discuss this issue in Section 8.2.

3.3. Possible Attacks

According to the characteristics of driver operations, we spell out attacks targeting at the I/O data. From the above typical I/O flow, we can find out many possible attack targets to get the value of the I/O data. We summarize all attack targets into three categories: device I/O interface, the kernel space data buffer, and the user-space data buffer.

— *Attacks on Device I/O Interfaces.* The device I/O interfaces include PIO, MMIO, and DMA descriptors. For PIO and MMIO, a rootkit may launch the I/O-port or MMIO mapping attack [Zhou et al. 2012] to intercept or manipulate the device I/O, or directly access the interface to get the I/O data. It may also attempt to modify the DMA descriptor to induce the device sending or fetching the I/O data to or from the memory regions controlled by the rootkit.
— *Attacks on Kernel Space Data Regions.* This type of regions include all driver allocated memory regions. A rootkit can keep probing and reading the target I/O data, or be triggered by some special event (e.g., external interrupt) to access the I/O data directly from these regions. It is hard for the kernel to defend against such attacks because the rootkit has the same privilege as the kernel. Another attack is that the rootkit calls the driver's legitimate routine to access the data.
— *Attacks on User Space Data Regions.* The user space regions are wildly open for a kernel rootkit. Once the I/O data is in the user space, the rootkit is able to bypass kernel- and user-level protections to directly access it. Such attacks can be defended by serval user-space protection approaches [Chen et al. 2008; Yang and Shin 2008].

3.4. Security Requirements

Given that the locations of the I/O data can be categorized into two types: device interface (including PIO, MMIO, and DMA) and main memory, we summarize all required security properties of the I/O flow protection on each of them.

For the data in the device interface, we require that malicious code can not access or manipulate the data. The security properties are stated as follows.

— *SP*0. The physical addresses and I/O ports of all device interfaces cannot be updated once they are fixed by the BIOS during the system boots up.
— *SP*1. Any access on the data or to update data transfer parameters through the device interface should be intercepted and verified.
— *SP*2. Only accesses from trusted code blocks are granted.

[3]To avoid increasing the TCB size, we suppose that the platform administrator signs every driver to be installed in the platform. The hypervisor is pre-configured with the administrator's public key.

For the data in the main memory, we require that only the trusted code blocks are able to *read* the protected I/O data, and the executions of the trusted code blocks are protected. The security properties are summarized as follows.

— *SP*3. If the I/O data in a memory buffer is readable (plain text), access control must be enforced and only granted the trusted code blocks can access it.
— *SP*4. If the I/O data in a memory buffer is unreadable (cipher text), any code blocks are able to access it without triggering any verification mechanism.
— *SP*5. If a trusted code block is interrupted to give up CPU during its execution, its execution context must be saved and restored when it occupies CPU again.

3.5. Challenges

We now discuss the challenges in designing a system that provides the guarantee of confidentiality of the I/O data over the lifetime of the system. The first challenge is the complexity of the I/O sub-system. Different devices have different interfaces to communicate with the system. For instance, cameras use USB interface while the PS/2 keyboard is attached to the system with the PS/2 interface. Furthermore, for the same device with the same version, different platforms (e.g., Linux or Microsoft Windows) have different driver implementations. The diversity and complexity dramatically increase the difficulties to build a generic solution to protect all I/O flows.

The second challenge is the complex and intensive interactions between drivers and the kernel. Most driver functions are dependent on the kernel exported functionalities. For instance, the driver memory allocation and deallocation are heavily dependent on the kernel memory management component. The heavy dependence and intensive interactions make it extremely hard to distinguish if an access on the protected I/O data is driven by the benign driver or by the compromised kernel.

The third challenge comes from the power of attackers. Once attackers compromise the kernel, they are able to gain the kernel (highest) privilege, which allows attackers' code to freely access any memory regions and I/O ports. On the other hand, it is hard for the buggy monolithic kernel to completely defend against software attacks due to the large size and numerous attack surfaces.

4. DESIGN RATIONALE

A straightforward approach is that the hypervisor arbitrates whether a control flow can access the I/O data. It requires the hypervisor to introspect driver operations, which is difficult to implement due to the semantic gap (e.g., lack of details of driver operations) between the hypervisor and the driver. Considering the complexity of I/O operations, the workload on the hypervisor will inevitably expand its code size, and may significantly downgrade the whole system performance.

Isolation is a widely used method to protect program executions. To apply isolation on I/O data protection, one may propose location isolation or execution isolation. Location isolation is to place device drivers and the kernel's I/O subsystem into a separated domain, for example, a driver domain or Dom0 in Xen, or the hypervisor's space, for example, VMware, so that malware in the guest kernel cannot attack them directly. These approaches are efficient in terms of I/O performance. Nonetheless, the resulting protection is weak because the TCB size is increased significantly due to the drivers and the I/O subsystem.

In the execution isolation, the device drivers still reside in the untrusted guest kernel while their executions are escorted in a secure environment established by the hypervisor, similar to TrustVisor [McCune et al. 2010] and Overshadow [Chen et al. 2008]. The generic execution isolation is not applicable for I/O data protection, because I/O operations are featured with frequent hardware interrupts and intensive

driver-kernel interactions. Note that if the I/O subsystem is also enclosed in the execution isolation, it suffers from the same drawback as in the location isolation approach.

We adopt the idea of execution isolation, however, at a micro-level. It is well known that most of the driver code is for housekeeping purposes, such as error handling, resource allocation and cleaning up [Ganapathy et al. 2008], with only a small portion dealing with I/O data transferring. We further observe that among the code for data transferring, only a few code blocks, for example, an encoding function, need to process the I/O data, while the majority of them just move the data from one memory location to another without necessarily knowing the content. Based on these observations, we design DriverGuard as a fine-grained I/O protection mechanism, which enforces access control on the device interfaces and encrypts the I/O data once it is moved into memory. To let the device driver work properly, DriverGuard distinguishes those security-sensitive driver code (around 1% of the driver code according to our experiments) from the rest. Only these identified code blocks are granted to access device interface, and access decrypted I/O data. Other code blocks is only able to access encrypted I/O data. In the meantime, DriverGuard protects the execution of security-sensitive code block to prevent malicious code from accessing the I/O data. Different from the hypervisor introspection technology, those access controls do not impose comprehensive semantic logics on DriverGuard. Hence, its performance is on par with the location isolation solution, however, the security strength is much stronger.

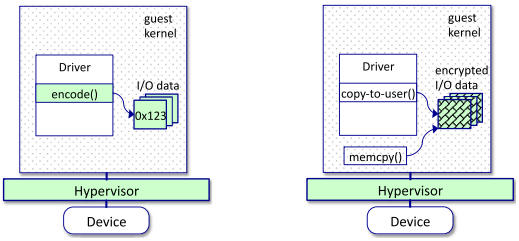
5. DESIGN OVERVIEW

By and large, DriverGuard is constructed using three lightweight protection techniques as the building blocks: cryptography, access control and runtime protection. We use cryptographic techniques to protect all I/O data without interfering with most of the driver and the kernel executions. For regions holding data that cannot be protected by encryption, we resort to DriverGuard to enforce access control. The plaintext data can only be accessed by a few designated driver code blocks, which are trusted and whose executions are safeguarded by our runtime protection mechanism. We refer to these code blocks as *privileged code blocks* (PCBs) in the rest of the article. By protecting the execution of PCBs, we successfully ensure the whole I/O data security with minimal overhead since PCBs only constitute a tiny fraction of the driver code.

5.1. Protection Mechanism Overview

A high-level view of DriverGuard's protection mechanism is as follows. Once the hypervisor boots up, it fixes all physical addresses and I/O ports of all device interfaces by setting read-only on the configuration space registers, and rejects any update requests from the guest OS [Zhou et al. 2012] (achieve $SP0$). All device interfaces related to protected I/O flow are enforced access control by the hypervisor, so that any access from the guest must be trapped into the hypervisor (achieve $SP1$). If the access is from a PCB (i.e., command-PCB or computation-PCB), the hypervisor grants the access, otherwise rejects it (achieve $SP2$). Receiving the I/O data from device interface, a PCB may attempt to read the content of data to do some computations, such as encoding or decoding operations. Before the PCB is off the CPU, the PCB is designed to either require the hypervisor set access control back on the data if it is readable[4] (achieve $SP3$) or encrypt the data into cipher text with the key (achieve $SP4$, and see more in Section 5.4), which is generated by the key-PCB and only accessible by PCBs. Non-PCBs are free to move the ciphertext to anywhere without any constrains from the hypervisor. Figure 2(a) and Figure 2(b) illustrate the difference between a PCB's and a non-PCB's

[4]The PCB is able to get enough semantic information to know if the I/O data is readable or not.



(a) A PCB, e.g., *encode* function, accesses the I/O data in plaintext.

(b) A non-PCB, e.g., the *memcpy* function, accesses the encrypted I/O data.

Fig. 2. The concept of privileged code block (PCB).

I/O data accesses. Since the PCBs never actively give up the CPU until its execution flow ends, the off-CPU event must be triggered by external interrupts or exceptions. Based on this observation, the hypervisor enables interception mechanisms on all interrupts and exceptions during the PCB execution. If the interception mechanism is triggered, the hypervisor restores protection on the I/O data. Furthermore, the hypervisor also protects the PCB execution context to avoid indirect data leakage (achieve *SP*5).

In Section 3.2, we assume that every driver is signed by a trusted entity such as the platform's administrator. To ensure the initial integrity of the PCBs in a driver, we further assume that they have been explicitly labeled in the driver code before being signed and installed. Therefore, the signature on the driver code also ensures the integrity of PCBs. (We will discuss PCB identification methods in Section 6.1.) Next, we explain the design details of three building blocks and leave the discussion of their integration in Section 7, since it involves the details of I/O operations.

5.2. Access Control Over Critical Regions

Since we do not rely on encryption-capable devices, encryption is not applicable for data used by the hardware. To cordon off illicit accesses to the data, we utilize the hypervisor's access control mechanism. In general, the data regions are classified into memory regions and I/O ports, for which we apply different access control methods by leveraging the hardware features and the virtualization techniques available in the platform.

To intercept accesses to a protected memory region, DriverGuard sets the attribute bits in the corresponding Page Table Entries (PTEs), clears the corresponding IOPL bits, and sets up the I/O bitmap to intercept accesses to an I/O port. Note that the protected memory addresses are machine addresses not guest physical addresses (or named pseudo physical addresses). We use *checkpoints*[5] in the rest of the article to refer to both the IOPL bits and the PTEs marked by the hypervisor for the purpose of access interception. Although the aforementioned protection techniques are used in many existing schemes, for example, Chen et al. [2008] and Payne et al. [2008], we are confronted with two new problems. First, given a memory buffer, the hypervisor must make sure that the kernel cannot bypass the checkpoint to access the region, which is challenging for memory regions allocated by the kernel. Second, the hypervisor must ensure that the sensitive I/O data is indeed placed in the region with a checkpoint. The

[5]Our definition of *checkpoint* has no relation with the *checkpoint* for rollback in distributed systems.

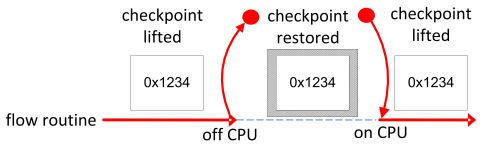


Fig. 3. An illustration of runtime protection, where 0x1234 is an exemplary memory address with a PTE checkpoint.

first problem demands a careful page table walk checking while the second demands the I/O control integrity checking.

5.3. Cryptographic Components

We introduce to the device driver a symmetric-key encryption function and a decryption function, both of which can be called by any code. However, any write access to the function code is denied by the hypervisor. We also add a key generation function to the driver as a PCB. The security of the I/O data relies on the secrecy of the driver's key, rather than the secrecy of the decryption function, which complies with the famous Kerckhoff's principle. The driver's secret key is securely generated based on a secret random seed supplied by the hypervisor. The secret key is securely stored in a kernel space buffer priorly appointed by the driver and can only be accessed by the driver's PCBs. This prevents any unauthorized code from decrypting the driver's data, even though the decryption function can be called arbitrarily.

5.4. PCB Execution Escorting

The third building block in DriverGuard is the runtime protection mechanism that prevents a PCB's execution from deviating its expected behaviors. The protection is requested at the PCB's entry and is relinquished at the exit via hypercalls. The hypervisor agrees to admit a control flow into the escorting only when the request is issued from the driver's PCB, and agrees to discharge a flow from escorting only when the request is issued from the PCB presently under escorting. The PCB is registered to DriverGuard during the kernel boot up process (details in Section 7).

The PCB under the escorting is granted by the hypervisor to access the critical data such as the driver's secret key and the I/O data, or to issue I/O commands. In our design, the hypervisor temporarily restores the access on those regions for the PCB, and withdraws the access right at the exit of escorting. Therefore, no duplicated exceptions or page faults will be raised despite that the PCB may access the same region multiple times within one escorted execution. An escorted PCB can be scheduled off from the CPU for various reasons. In that case, the hypervisor intercepts these events and restores all checkpoints. Meanwhile, it also securely saves the driver's runtime stack and sets up a breakpoint for the PCB's upcoming CPU occupation. As a result, other code's accesses to the protected regions are denied. Figure 3 depicts a scenario of escorting.

6. PRIVILEGED CODE BLOCK

We consider three types of PCBs in a driver. One is *computation-PCBs*, which refers to the driver code blocks making computation on the I/O data, for example, an encoding function. The second is *command-PCBs*, which refers to the driver code blocks issuing data transfer parameters to the device. This type of code is security sensitive because their executions determine the locations of plaintext I/O data. The third is *key-PCBs* which refers to the driver code blocks initializing the driver's encryption key. Each driver generates its own key in the driver initialization step, such as in `module_init`.

There are several properties of PCB summarized as follows: (1) It is self-contained in the sense that there is no extra function calls to kernel functions; (2) It does not

Fig. 4. The two types of PCB format. (a) A PCB ending with encryption. Thus, the hypervisor does not need to enforce access control on the data; (b) A PCB ending with protection requirement. Therefore, the hypervisor must enforce access control on the data to restrict the access.

contain indirect call or indirect jump (e.g., no call using function pointer), meaning that the control flow is static; and (3) It does not have any dynamic data dependence except for the parameters. These properties call for driver developers' prudence in driver coding such that the driver code is friendly to PCB identification.

6.1. Identifying PCB

Given that the *key-PCB* is added by the DriverGuard scheme, we only illustrate how to identify the other two types of PCB from the driver code. Following the I/O data flow, the sophisticated driver developers are able to identify all functions that operate on the I/O data, and thereby label all PCB candidates in these functions. If some PCB candidates are not naturally satisfy the above listed PCB properties, there are some guidelines for the driver developers to modify them into PCBs.

Obviously, it is easy to achieve the second property by carefully programming. To achieve the first property, the developer can replace external function calls with its own code if they are simple (like *inline* functions). If they are hard to be replaced, the developer may either move these function calls out of the PCB candidate if they do not effect the behavior of the driver, or divide the PCB candidate into two PCBs with the function call as the separator. In order to achieve the third property, the developer could assign the dynamic dependence data to the static or global data variables. To facilitate the protection on these variables, developers are able to put them in a particular region (e.g., a pre-reserved page) with compiling flags. Note that the labeled driver can be distributed after the PCB-labeling work is done.

6.2. PCB Format

A PCB is always capsulated by a pair of hypercalls for escorting. The entry hypercall is a *start_escort* hypercall that requires DriverGuard to start to protect the execution of the PCB, and the exit hypercall is a *end_escort* hypercall that informs DriverGuard to end the PCB execution protection. There are two possible formats for a PCB. One format of the PCB is ended with encryption on the protection data. Such PCBs are usually in the intermediate parts of a driver, where the driver does some process operations on the I/O data whose output is ready for the later stage. The other is ended with protection requirement which is to request DriverGuard to block all accesses on the protected regions. Such PCBs usually directly work with device (e.g., updating I/O buffer for DMA transferring) or user-level applications (e.g., copying data into user space). We illustrate their different formats in Figure 4.

We assume that the PCB interface is trusted and well-designed/implemented, without malicious intention to leak I/O data to outside. In fact, it is true in almost all cases especially for the ones in the driver interfaces since they are usually well defined in the specification.

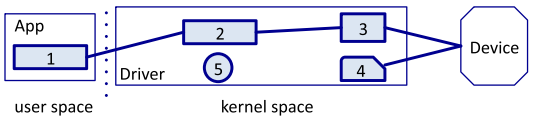


Fig. 5. An illustration of five types of regions with same numbering in the description.

7. DESIGN DETAILS

We build DriverGuard on top of the Xen hypervisor to protect the drivers running in a Linux guest domain. We systematically examine every step in I/O operations, from the device discovery to the application's (or device's) data fetching. In order to adaptively protect the driver operations, the hypervisor needs to store certain context information about the driver. We start with driver context initialization since it is performed by the hypervisor during the guest domain bootstrapping.

7.1. Driver Context Initialization

The context information of drivers are securely stored in three types of tables in the hypervisor space. A *device table* specifies the management relation between a driver and a device by paring their identifiers. For every protected driver, the hypervisor maintains a *PCB table* and a *region table*. The former stores the entry and exit addresses of all PCBs of the driver while the latter specifies the memory regions and the I/O ports to protect. There are five types of regions in the region table: (1) the application buffer; (2) the memory buffer allocated by the driver for data processes; (3) the I/O data buffer such as DMA buffers; (4) the device interface, including the I/O ports or MMIO regions and DMA descriptor queues; and (5) the buffer holding the driver's secret key. Figure 5 depicts their locations in the system.

Device Table Initialization. When a guest kernel image is uncompressed, the hypervisor inserts a hook function to the kernel to inform the hypervisor about the device-driver association via a hypercall. The hypervisor then initializes the device table accordingly. The hypervisor also sets the checkpoints for the kernel structure maintaining the device-driver association. Whenever a driver takes the ownership of a device, the hypervisor intercepts the event and updates the device table properly.

PCB Table Initialization. We assume that all PCBs in a driver have been manually identified and delimited by a pair of hypercalls, that is, an escorting-entry hypercall and an escorting-relinquish hypercall. The hypervisor scans the driver code to record the addresses of escorting-entry hypercalls and of the respective escorting-relinquish hypercalls. It puts these pairs into the PCB table. In Section 8.1, we will discuss how to automatically identify PCBs.

Region Table Initialization. The regions used by a driver can either be the default ones chosen by the manufacturer/the kernel or set by the driver. In the first case, the hypervisor updates them when the driver is loaded as in the device discovery step. In the latter case, the driver informs the hypervisor via a hypercall about the protected regions or I/O ports.

Driver Key Initialization. Each driver has a dedicated key-PCB to initialize its own encryption key. In the key-PCB, key initialization algorithm first issues a hypercall to get a random seed from hypervisor, and then saves the generated key into its key buffer. The key generation process is only run once, and escorted by hypervisor (see escorting details in Section 7.3.2).

7.2. Checkpoint Deployment

Given a memory region or an I/O port, the hypervisor sets up the corresponding checkpoint to intercept and verify potentially malicious accesses. The detailed deployment method is dependent on the virtualization environment.

7.2.1. Memory Region Checkpoint. For a memory page *A*, the hypervisor walks through the page tables through the *CR*3 register to locate the corresponding PTE pointing to it. The hypervisor sets the attribute bits on the PTE to specify different access rights. To set a page *read-only*, the _PAGE_RW bit is cleared; and to set a page *nonaccess*, the _PAGE_PRESENT bit is cleared. Note that all protected regions are in kernel space and all processes share one kernel space mapping. In the paravirtualization setting, only the hypervisor is able to update page tables. In the hardware-assisted virtualization setting, a Shadow Page Table (SPT) or Extended/Nested Page Table (EPT/NPT) is used to translate virtual addresses into machine addresses, and the SPT/EPT/NPT is only updated by the hypervisor. Although the mechanism of the SPT is a little bit different from the one of EPT/NPT, that is, the SPT requires the hypervisor to control over the guest page table while the EPT/NPT do not need, the page-access-checking mechanisms are essentially the same, since the final access right to a page is determinate by the page table, that is, SPT/EPT/NPT, handled by the hypervisor. Therefore, those checkpoints can not be removed by the malicious kernel. Note that in the hardware-assisted virtualization environment, the hypervisor enforces access on the machine address, not the pseudo physical address or virtual address.

Given that the granularity of the memory protection is in the page-level, we should carefully deal with the I/O data buffers that are not page-aligned or mixed with other data in a single page. There are two options to solve the problem. One is to request I/O buffer at the length of pages. In fact, to accelerate the performance, many device drivers have such allocation feature. For example, the USB camera driver allocates a large memory pool in page level for data caching. The other option is to let the hypervisor emulate the operations that access other data. In DriverGuard, we choose the second option to avoid changes on driver code. Although emulation incurs performance loss, the likelihood of its occurrence is low. This is because the checkpoints are only deployed on device interfaces used immediately after or before I/O, and other data are protected by encryption.

Legitimacy of Memory Region. When the hypervisor attempts to set up a checkpoint, it checks whether the machine memory page can be reached by another unauthorized PTE. In other words, the kernel is not allowed to bypass the checkpoint to visit a machine memory page. Therefore, the hypervisor must ensure that there exists no unchecked virtual-to-machine address translation for memory pages with checkpoints.

We leverage the hypervisor's memory management mechanism to tackle this issue. The Xen hypervisor maintains a `page_info` structure for every machine memory page. The `count_info` field in this structure records the number of usages of a machine memory pages. For a page allocated to a guest, its `count_info` is actually 2, because the hypervisor itself is holding it.[6] Therefore, on setting up a PTE checkpoint, the hypervisor checks if the corresponding counter is 2. In addition, we modify the hypervisor's `do_mmu_update` and `do_update_va_mapping` to prevent the kernel from crafting a trapdoor path for existing checkpoints. These functions are used by the guest kernel to update page tables. In this way, for any page table update, the hypervisor checks whether

[6]Before a machine memory page is allocated to a guest OS, the hypervisor holds it first and sets the `PGC_allocated` bit. Therefore, the page's `count_info` is already 1 before being allocated to the guest.

Admission Algorithm:

1) Fetch the EIP value stored at the top of the current guest kernel stack, which is the return address of the hypercall.
2) If EIP does not match any entry in the PCB table, return error.
3) If the address of requested buffer is legitimate, then
    a) set InEscorting to 1;
    b) If the guest's kernel stack segment is not a dummy stack, then
        (i) allocate a dummy stack at the reserved space.
        (ii) save the machine addresses of the dummy stack and the present stack as (MA'_{ss}, MA_{ss}). Return 0.
    c) else, switch to the corresponding genuine stack. Return 0.
6) Return -1 as an error message for admission failure.

Fig. 6. Algorithm for PCB admission.

the requested update increases the usage counter of any machine memory page with a checkpoint.

In the hardware-assisted virtualization environment, the hypervisor does not set two PTEs pointing to the same machine address. To enforce this property, the SPT/EPT/NPT update algorithm can be extended to verify it. Note that the legacy hypervisors, such as Xen, do not export any interface for the guest to manage the SPT/EPT/NPT.

7.2.2. I/O Port Checkpoint. I/O ports is separated from the memory address space, and the accesses to such I/O ports need a set of special instructions, for example, *inb* and *outb*. A successful access must go through the IOPL checking and I/O bitmap checking. Any access will be blocked once its priority is lower than the priority specified in the IOPL. Even if the access pass the IOPL checking, it is still blocked if the corresponding bit is set in the I/O bitmap. To prevent the malicious guest kernel from accessing the protected I/O ports, the hypervisor clears the IOPL bits of *EFLAGS* of the guest's CPU. Namely, it sets the I/O privilege level to 0, such that the hardware always checks the I/O bitmap for PIO instructions because the paravirtualized kernel runs in Ring 1. Then, the hypervisor sets the bits corresponding to the protected I/O ports such that a PIO instruction will cause a general protection exception.

It is relatively easier to set up I/O checkpoint in hardware-assisted virtualization. More specifically, the hardware-assisted virtualization technique supports that the hypervisor itself can intercept all instructions that access a particular I/O port through a dedicated I/O bitmap in the hypervisor space. Therefore, the hypervisor simply activates the I/O bitmap mechanism by setting the 25th bit of the processor-based VM-Execution control vector and then sets the bits in the bitmap corresponding to all protected I/O ports.

7.3. PCB Execution Escorting

7.3.1. PCB Admission. A driver's PCB starts with the hypercall which takes as the parameter the buffer address it requests to access. To admit a PCB, the hypervisor checks whether the hypercall is issued from the instruction whose address is registered in the PCB table. If not, the hypervisor rejects the request.

For an admitted PCB, the hypervisor protects its stack as follows. The hypervisor allocates a dummy stack for the PCB. Therefore, an admitted PCB has two runtime stacks. A genuine stack is used for the PCB's execution while the dummy stack is used for untrusted code sharing the same execution flow due to interrupts. The usage of dummy stacks will be explained in the next subsection. Figure 6 describes the details of the PCB admission algorithm, where InEscorting is a flag bit indicating the current execution state.

IRQ-handler Algorithm:

(1) If $InEscorting = 0$, return.
(2) Restore the checkpoints that are removed during escorting.
(3) Switch to the dummy stack, by setting the PTE for the guest's stack base to point to MA'_{ss}.
(4) Set $InEscorting = 0$.
(5) Set a local breakpoint at the instruction pointed by EIP. Save the address pair in EIP and ESP.
(6) Return and pass the control to the default interrupt handler.

Fig. 7. Interrupt handler for escorting. When there is an interrupt interrupting the execution of the execution of a PCB, the interrupt handler restores the protection on the escorted data, and saves the context of current escorting PCB.

7.3.2. Escorting. Once a PCB is admitted by the hypervisor, its execution is escorted and the checkpoints for the buffers are temporarily lifted. The essence of escorting is that the hypervisor intercedes whenever the PCB is scheduled off from the CPU, which occurs due to the hardware interrupts. This situation opens the door to the kernel attacks, because the kernel may occupy the CPU and could access the PCB's runtime stack and data. To defend against such attacks, the hypervisor should be able to enforce access control on the data and stack before the potentially malicious kernel occupy the CPU. In the virtualization environment, the hypervisor is able to configure the system to give priority to itself to occupy the CPU. Specifically, all hardware interrupts are sent to the hypervisor prior to sending to the guest domain. Therefore, the hypervisor is able to (1) restore the checkpoints and (2) replace the runtime stack with the dummy stack allocated in PCB admission. The hypervisor also sets a breakpoint to intercept the events that the PCB is rescheduled to the CPU.

We explain here how the hypervisor handles an interrupt through interrupt handler `do_IRQ` and a debug exception through the debug exception handler `do_debug` in addition to its normal process.

Interrupt. To switch to a dummy stack, the hypervisor only replaces the content of the PTE for the present stack with the machine page number of the dummy stack allocated during admission. This change is transparent to any guest process, since the address in the *ESP* register remains the same. Hence, the guest kernel is not able to access the true stack while the subsequent execution can use the dummy stack without being affected. The algorithm for stack switching and checkpoint restore is shown in Figure 7.

Debug Exception. When a debug exception occurs, the hypervisor's *do_debug* function is called before the event is forwarded to the guest kernel. All breakpoints used by the hypervisor are local breakpoints. Therefore, they are triggered only for the present process. There are two types of local breakpoints used in DriverGuard.

Setting a breakpoint at the *EIP* is to intercept the event of PCB resuming. For this type of breakpoint, the hypervisor enters into escorting only when both *EIP* and *ESP* values match the previously saved pair. The details are shown in the following algorithm in Figure 8.

7.3.3. PCB Exit. To exit from the hypervisor escorting, the PCB issues another hypercall. The hypervisor checks if InEscorting is set. If not, it returns an error message; otherwise, it clears InEscorting flag. The PCB should also issue a hypercall to protect its data if the data are left in plaintext. The hypervisor sets no more breakpoints and processes interrupts and exceptions in the normal way.

Debug-handler Algorithm: Breakpoint address stored in *EIP*, the stack address stored in *ESP*

/* Enter into Escorting */
(1) If there exists a saved (EIP', ESP') pair, s.t. $ESP' = ESP$ and $EIP' = EIP$, then
  (a) remove the breakpoint at *EIP*;
  (b) Restore to the genuine stack by replacing the stack PTE with MA_{ss}.
  (c) Set $InEscorting = 1$, and return 0.
(2) Return -1 as an error message.

Fig. 8. Exception handler for escorting. When a previous interrupted PCB resumes, the exception handler restores the PCB execution context.

7.4. Data Region Access Control

A potentially malicious access to a memory region with a checkpoint causes a page fault and an access to an I/O port with a checkpoint throws out a general protection exception. Therefore, we modify the hypervisor's page fault routine `do_page_fault` and the general protection exception handler `do_general_protection`. In the former, the hypervisor gets the address of the trapped instruction from EIP and the address being checked from CR2, while in the latter, the I/O port number is enclosed in the instruction.

If the access is granted by the hypervisor, the event will not be forwarded to the guest kernel. In that case, The legitimate flow continues to execute the intercepted instruction without being re-scheduled due to the page fault as the guest kernel does not observe this exception. For unauthorized accesses, the page fault or exception is passed to the guest kernel. DriverGuard is compatible with memory mapping for page sharing because the checkpoints are deployed at the PTEs. A buffer mapped to two addresses has two PTE checkpoints. In the following, we elaborate the details of region access control according to all types of regions except the control region.

I/O Buffer. The addresses of I/O buffers are obtained within an escorted command-PCB. Since the I/O buffer contains the data to/from the device, they are not protected by encryption. The hypervisor blocks all accesses not from an escorted PCB. For an input buffer containing the data from the device, the driver always encrypts the data before moving them to other locations, whereas for an output buffer the driver must decrypt the data after copying them to the output buffer.

Driver Buffer. Driver buffers temporarily hold data for processing. When the data in those buffers are encrypted, the hypervisor does not set up checkpoints for them. Only when the escorted PCB is temporarily scheduled off from the CPU, the hypervisor sets up the checkpoints against all accesses as the data are in plaintext. In this case, the PCB notifies the hypervisor about the buffer address.

Key Buffer. The key buffer holds the secret encryption key used by the driver. The hypervisor allows the key to be read only from the instructions from the encryption/decryption functions and is currently in escorting mode. Thus, a non-PCB can not access the encryption key.

7.5. Device Control Protection

As explained in Section 7.1, the control region's addresses can be obtained in the initialization phase for certain devices. The hypervisor denies all write accesses to the region not from an escorted PCB. Furthermore, it is also crucial to maintain the consistency between the I/O buffer address specified in an I/O command that is sent to the control region and the buffer addresses requested by the device driver. This is because the kernel may manipulate the I/O command such that the device uses an unprotected I/O buffer for transferring. To defend against such attacks, the driver's command-PCB

informs the hypervisor the locations of the I/O buffers in use, such as the DMA buffer and the DMA descriptor queue. The hypervisor inserts them in the region table and sets up the checkpoints accordingly. Therefore, it ensures that the I/O buffer in use is always protected.

7.6. Device Configuration Space Restriction

In order to defend against I/O-port and MMIO mapping attacks, DriverGuard restricts the updates on the physical addresses and I/O ports of device interfaces. According to the descriptions in Section 2.1, DriverGuard sets checkpoints on the I/O ports 0x*CF*8 and 0x*CFC*, or the reserved memory region. The details of the checkpoint refers to Section 7.2. Any update (write) operations will be rejected. This restriction does not lower the runtime performance of the system since the configuration operation is normally done once at the bootup phase of the system.

7.7. User-Space Device Driver Support

When a device is managed by a user-space driver, the I/O data is directly transferred between a user space buffer and the device interface without any intermediary kernel space buffers. According to our threat model in Section 3.2, the user space memory regions are well protected by schemes like Overshadow. However, those schemes are not sufficient for I/O protection because they do *not* protect the device interface. Thus, we need to instrument the driver code with hypercalls to fix the protection gap. The inserted hypercalls update the information of the device interface to DriverGuard and request it to enforce access control on the device interface. Recall that all the user space driver code is protected. Thus, we do not need to identify PCBs for user-space drivers.

8. DISCUSSIONS

In this section, we discuss the automatic PCB identification process, and the whole lifecycle of protection on the I/O flows with the cooperations of DriverGuard and other user space approaches.

8.1. Automatically Identifying PCB

Ideally, a fully automated PCB identification algorithm can discover PCBs in a device driver with no false positives and no misses. False positives lead to unneeded overhead while misses result in loopholes for the adversary to attack. However, it remains as an open problem how to design such a PCB identification algorithm for a driver's source code or binary code. In this article, we make analysis of the challenges and propose a best-effort solution.

Recall that we defined three types of PCBs in Section 6. We only need to discover computation-PCBs and command-PCBs, as the key-PCBs are new function inserted to the driver. Labeling command-PCBs is straightforward, since they are featured with special instructions (e.g., *inb* and *outb*) involving the device interface (e.g., I/O ports) or special memory region (e.g., MMIO). Many existing techniques can be used to identify the related statements, for example, interprocedural points-to analysis technique [Heintze and Tardieu 2001] and the slicing techniques [Mock et al. 2002; Sridharan et al. 2007; Weiser 1979]. For instance, in the slicing techniques, we select the device interface (e.g., the I/O ports) as the seed, and the slicing tool can find out all statements that directly access the seed.

The computation-PCBs are the code blocks computing on the I/O data (e.g., mapping the scan code into key code in the keyboard driver). Identifying the computation-PCBs is a challenging task since it involves code and data semantics. To the best of our knowledge, existing code analysis techniques (e.g., forward and backward slicing, and

thin slicing) are not sufficiently intelligent to distinguish code semantics. Another challenge is the abundant usage of function pointers in drivers, which makes it infeasible to determine execution flows through a static code analysis. This issue is aggravated by the fact that most drivers are essentially a collection of disjointed functions, instead of a single executable. The executions of driver functions are usually integrated with kernel execution. It is therefore difficult to map out all possible execution flow given existing code analysis techniques.

We propose a semi-automatic method for computation-PCB identification, with automated tools for coarse-grained scope defining on a large scale of code and human efforts for fine-grained refinement on small scale code fragments. Given a driver's source code, which is a set of functions, the basic idea is to first pick up functions related to I/O data, then identify PCBs within each chosen function. In a nutshell, the procedure is divided into three steps: (1) to select functions that potentially contain PCBs; (2) to identify PCB statements in each function selected in previous step; and (3) to form all PCB blocks from the statements discovered in Step 2.

Step 1. Function Candidate Selection. Drivers are highly structured code to conform to hardware interface specifications, such as providing file operation interfaces like `open`, `read`, and `close`. According to hardware specifications, I/O flows must originate at those designated interfaces. Therefore, those interface functions that do not process I/O data, as well as functions solely called by them, are excluded from our search scope. For instance, the interface function *poll* corresponding to the *select* system call usually does not access the I/O data, while allows a program to monitor and wait until one or more of driver/device states become ready for some class of I/O operations. The interface functions like *read* and *write* usually handle I/O data for the requests of the applications. For ease of presentation, we use $\mathcal{F}$ to denote the set of interface functions with I/O flows.

We then map out the execution flows (and therefore I/O data flows) starting from functions in $\mathcal{F}$, so that all dependent functions are examined. For this purpose, we first manually identify the I/O data used in $\mathcal{F}$, because the exact locations of I/O data in those functions are implementation specific, for example, in the function parameters or predetermined buffers. Then, we use the I/O data as the seed to perform dynamic taint analysis [Kemerlis et al. 2012; Newsome and Song 2005] to identify functions involved in I/O flow.[7] Since the dynamic taint analysis does not guarantee covering all execution paths, manual efforts are needed to check missed functions. Lastly, we extend $\mathcal{F}$ to enclose all functions picked up either manually or by the tool. For each function in $\mathcal{F}$, we identify computation-PCB in the next step.

Step 2. PCB Statement Identification. In each selected function, we attempt to identify from the function body the PCB-candidate statements where I/O data are involved. Using the I/O data in Step 1 as the seed, we apply the slicing tools [Mock et al. 2002; Sridharan et al. 2007] to label all seed-related statements in the function body. Note that the resulting statement set contains non-PCB statements for two reasons. First, according to Santelices et al. [2012], slicing techniques may introduce false positive in statement discovering. Second, it is likely that some statements correctly identified by the slicing tools are not for I/O data computation, since the slicing technique does not take code semantics into consideration. For instance, statements that copy I/O data between memory buffers do not satisfy the definition of computation-PCB. We suggest to manually examine the slicing results to filter out non-PCB statements.

[7]Although the method for dynamic taint analysis is applicable to drivers in principle, we have not found any existing tool suitable for this task.

Step 3. PCB Formation. The last step is to organize the PCB statements in Step 2 into PCBs and instrument them with hypercalls. If each PCB statement is treated as a PCB block, its performance toll will significantly rise up. For each function in $\mathcal{F}$, our algorithm scans the statements identified in Step 2 with several rounds of iterations. In the first iteration, adjacent PCB statements are grouped into one PCB block. In the second iteration, the algorithm attempts to merge separated PCBs in order to reduce the total number of PCBs. If two PCBs are in the same basic block (i.e., a straight-line sequence of code with one entry point and one exit) and the number of non-PCB statements between the PCBs are less than a predetermined parameter κ, then these two PCBs are merged together with the non-PCB statement in between into a new PCB. Note that κ is used to tune the balance between the size of PCB and the number of PCBs. This iteration continues until no new PCB is generated. In the end, two hypercalls are inserted for each formed PCB as described in Section 6.

It is better for the driver developers to do the PCB identification since (1) they know best, and (2) it may increase the market share due to the extra security services. In addition, the identification process is only done once, and the results can be delivered anywhere. For a particular system, the hypervisor does not need to maintain a universal list (including all PCBs), while it manages the PCB list only for the drivers loaded in the system. Maintaining the PCB list that are never used will lead to unnecessary cost and may increase the TCB size, especially for the hypervisor.

8.2. Full I/O Path Protection

As illustrated in Figure 5, a full I/O path in general consists of the user-space buffers allocated by the application, the kernel-space buffers allocated by the driver and/or kernel, and the I/O interface buffer such as a DMA buffer. DriverGuard ensures the security of the latter two while a user-space protection scheme (e.g., Overshadow [Chen et al. 2008], SP^3 [Yang and Shin 2008] or SecureME [Chhabra et al. 2011]) secures the first type of buffers. To have a seamless integration, the key issue is to ensure that the kernel segment of the I/O path correctly joins the intended application's user space segment, as a device is shared among multiple applications.

This issue has twofold implications. One is that the location of the user space buffer allocated by the target application must be securely passed to the driver, such that the driver can deliver (fetch) I/O data to (from) the right place. The other is that the data during the user-kernel space transition must be securely handled. When the I/O data is transferred between the application's buffer and a kernel buffer, it should be ensured that no security gap exists during the transition. In other words, both buffers should be protected either by encryption or hypervisor-based access control while allowing data flow between them.

We propose here a design integrating DriverGuard with Overshadow as illustrated in Figure 9. Note that Overshadow makes use of a cloaked shim that is introduced as a trusted component in user space. The shim code in Overshadow plays the role of protecting data exchange between the application and the kernel using system calls. We propose to add a new function named as *shimguard* in the cloaked shim. Shimguard in the cloaked shim handles inbound and outbound data before and after system calls and correctly locates the application buffer for I/O data.

Before an application issues a system call to activate an I/O operation, the shimguard is involved by the cloaked shim. The shimguard first updates the identity of the application, the identity of the buffer and the identity of the target driver to the hypervisor. The identity of the application is the unique *address space identifier* (ASID) maintained by the hypervisor as proposed in Overshadow. If current ASID is the trusted application, the hypervisor accepts the hypercall; otherwise, it will reject

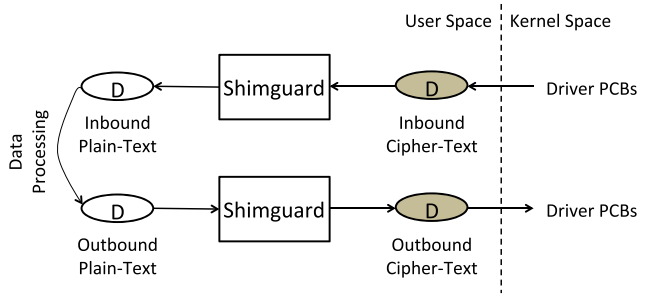


Fig. 9. The *shimguard* helps Overshadow and DriverGuard to protect the whole life cycle of the I/O data. Note that the I/O data denoted as *D* in the shaded regions are encrypted either by driver PCBs or by shimguard.

the hypercall. The identity of the buffer is its memory region represented in machine address. The start and end virtual addresses of the buffer are provided by the shimguard using the hypercall, and the corresponding machine addresses are collected by the hypervisor. Note that the virtual addresses alone can not used as the identity of the buffer since they may represent a different buffer in another virtual pace. The identity of the device driver can be got by using the name of the device provided through the hypercall since the device and driver mapping relationship is maintained by the hypervisor. The hypervisor generates a unique AES encryption key for the received 3-tuple of identities. Both the 3-tuple and the corresponding key are inserted into a table in the hypervisor space. For clarity purpose, we use *shim-key* to denote the AES key save in couple with the 3-tuple identifiers.

We use a read operation as an example to illustrate how to protect the I/O path starting from the device interface to the application buffer. The protection over the kernel space segment is the same as described in previous sections. When the PCB of the driver moves the I/O data into a user space buffer denoted as *Addr*, it requests the shim-key for *Addr* from the hypervisor. The hypervisor releases the shim-key on the condition that there exists an entry in the previous table containing both the requesting driver's identity and the machine address of *Addr*. If successful, the driver PCB encrypts the I/O data and transfer the cipher text to *Addr*. When the cloaked shim is trigger to fetch the data from *Addr*, the shimguard function requests the shim-key from the hypervisor. The hypervisor releases it on the condition that there exists an entry in the previous table containing both the requesting application's ASID and the machine address of *Addr*. If successful, the shim deciphers the encrypted I/O data and passes it to the application.

The operations for outbound data flow is similar to this description. Note that the shim-key in different from the the encryption key used by the driver described in previous sections. The shim-key is application specific and is the same as that used in Overshadow, whereas the driver has its own encryption key.

9. EVALUATION

We implement DriverGuard and run experiments on six peripheral devices to evaluate its security and performance. The devices are a USB keyboard, a web camera, a fingerprint reader, a sound card, a printer and a graphic card.

9.1. Security Analysis

9.1.1. Driver Security. As we know drivers are usually buggy. Attackers are able to compromise the driver through these vulnerabilities to hijack the control flow or data flow

of the driver to attempt to get the I/O data. Fortunately, attackers can not get the I/O data as long as the integrity of driver's PCBs are kept under the protection of the DriverGuard. Even if attackers completely control other parts of the driver, they are only able to access encrypted I/O data or are directly rejected since all these accesses are not from PCBs. Smart attackers may attempt to call a PCB to get I/O data. However, this attempt would fail because (1) the control flow of the PCB is static and (2) I/O data is either encrypted or set protection by the hypervisor according to the design of the PCB (Section 5.4) when the control flow is out of PCB . The confidentiality of the I/O data is dependent on the trustworthiness of PCBs, not other parts of the driver or kernel. Therefore, attackers cannot get any benefits from buggy device drivers.

9.1.2. ROP Attack. The Return-Oriented Programming (ROP) attack is a very powerful attack since it does not need to inject malicious code but drives legitimate code to do malicious behaviors. However, the ROP attack cannot get the protected I/O data in our system due to the design of the PCB and the DriverGuard protection. More specifically, in our design, the executions of PCBs are protected by the hypervisor and the control flows of PCBs are static. Therefore, attackers cannot hijack any PCB control flows. Furthermore, only the execution flows that start from *recorded start_escort* hypercalls are able to access decrypted I/O data. Any other execution flows that have no escorting request or with *unrecorded* requests are rejected.

9.1.3. DMA Attack. Our scheme relies on IOMMU to defend against DMA-based attacks, whereby a rootkit instructs a DMA device to read/write a memory region. IOMMU can defend against this type of attacks if the checkpoints are set on I/O page tables as well. If IOMMU is not available, an alternative approach is to intercept DMA request with shadow DMA descriptor mentioned in BitVisor [Shinagawa et al. 2009]. Nonetheless due to its high runtime cost, the shadow DMA descriptor is more amiable to slow devices with infrequent usage, for example, a fingerprint reader.

9.1.4. Interrupt Spoofing Attack. The interrupt spoofing attack are proposed in Zhou et al. [2012], which attempts to induce the device driver operating on incomplete or inconsistent data by processing spoofed interrupts. Obviously, the interrupt spoofing attack may lead to the device driver's misbehavior. However, it can not help attackers to access the I/O data, since it is only readable for trusted PCBs.

9.1.5. Side Channel Attacks. Side channel attacks (e.g., White et al. [2011] and Song et al. [2001]) can be used by the adversary to infer secret data. Since the adversary in our model refers to malwares residing in the guest OS, the hardware-based side-channels such as power consumption are not feasible for the adversary. The adversary can launch other attacks by observing the timing difference between two I/O events (e.g., keystrokes) or the contents in a CPU cache, which is weak than the adversary considered in chip-card security. In addition, existing side channel attacks mainly target cryptographic data, such as a decryption key or a password. It is unknown whether generic I/O data is subject to these attacks as well.

Our current design does not take side-channel attacks into consideration. To counter these attacks, DriverGuard should deploy a special AES implementation resisting side-channel attacks. The hypervisor should clean up the CPU caches whenever a PCB is scheduled off from the CPU. It can also generate random I/O events to defeat timing analysis. The main challenge is how to deal with side-channel attacks without increasing the hypervisor's complexity and incurring more overhead.

9.1.6. Attacks on Multicore Platform. On a multicore platform, it is possible that while a PCB runs in one core accessing the I/O data, the subverted guest kernel on another

core can also access them using the same page table used by the PCB. This attack can be countered using hardware-assisted virtualization supporting EPT or NPT.

The hypervisor prepares a dedicated EPTs for PCBs so that they have access permissions to those protected checkpoints. The non-PCB code such as the untrusted guest kernel use the normal EPT/NPT, in which the checkpoint regions are set as inaccessible. Whenever a PCB starts to occupy a CPU core, the hypervisor installs the dedicated EPTs (e.g., triggered by the *start_escort* hypercall) for the corresponding core. When it gives up the CPU core, the normal EPTs are restored (e.g., triggered by the *end_escort* hypercall). Since instructions on other cores do not have the dedicated EPTs, they cannot access the protected region when the PCB is in execution. Note that the EPTs are solely managed by the hypervisor. The guest kernel does not the privilege to manipulate the EPTs.

9.2. Security Evaluation

To validate the design of DriverGuard, we evaluate its effectiveness in several experiments.

9.2.1. Known Attacks. To the best of our knowledge, the only publicly known kernel-level attacks on I/O devices are keyloggers. We have downloaded and tested three kernel-level keyloggers and none of them can successfully acquire the keystrokes. The first keylogger[8] directly reads the keyboard I/O ports 0x0060 and 0x0064 using a fake interrupt handler. It fails because the fake interrupt handler does not belong to the authorized keyboard driver PCBs. Thus, it cannot access the I/O port or get the decryption key. The second keylogger[9] modifies the keyboard driver's data structure and installs a malicious function handler. Since the malicious function handler is not admitted by the hypervisor as a PCB, it can only access encrypted keystrokes without being allowed to use the secret key. The third keylogger[10] modifies the system call table to replace *read* function with a malicious one. The malicious *read* function first calls the original *read*, and then steals data from the user space buffer that is passed as parameter. This rootkit fails because the *read* function only copies the encrypted keystroke. After the driver places the ciphertext in the application buffer and opens it for the application, the hypervisor denies all kernel level accesses.

9.2.2. Synthetic Experiments. We introduce three synthetic attacks to read protected I/O data, and the results show that the DriverGuard successfully prevents all of them. More specifically, in the first experiment, we attempt to modify one byte in the protected MMIO region. DriverGuard catches the write operation through a page-fault exception. DriverGuard rejects the operation once it verifies the caught operation is not from an identified PCB. In the second experiment, we try to read the protected I/O data in a driver buffer, where the data is encrypted. We are only able to get the cipher text since the newly introduced code is not able to get the encryption key to decrypt it. In the third attack, we introduce a piece of code to call a PCB to get the protected I/O data. The attack fails as the data is encrypted when the execution flow is out of the PCB.

9.3. Usage of PCB

In our experiments, we manually identify all PCBs on the source code of device drivers and the drivers in the kernel's I/O subsystems, for example, a host controller driver. It

[8]http://www.phrack.org/issues.html?issue=59&id=14

[9]http://goo.gl/DpOBc

[10]http://packetstormsecurity.org/files/view/25677/kernel.keylogger.txt

Table I. Privilege Code Blocks

Driver	**Size** (LOC)	**# of PCBs**	**Avg. PCB Size** (LOC)	**Device**
keyboard driver	4964	11	17	keyboard
HID*	12771	13	10	keyboard
UVC driver	7838	7	11	camera
EHCI*	10011	6	15	camera
HDA-Intel	47825	8	6	sound card
Sound-core*	18722	5	4	sound card
devio	1628	7	12	printer, fingerprint reader
UHCI*	7600	5	14	printer, fingerprint reader

The number of PCBs and the average size for each driver used in our experiments. The drivers labeled with stars are those within the kernel's I/O subsystem. The PCB size includes the hypercalls and the calls to the encryption and decryption functions.

is straightforward to identify command-PCBs and key-PCBs, because key-PCBs are introduced by DriverGuard while command-PCBs are the code accessing port I/O, MMIO or structures used by devices (e.g., frame list of UHCI). Identifying computation-PCB requires the semantic knowledge of the code. We trace the I/O data to spot code segments computing on the I/O data. Note that code segments for copying or moving data are not PCBs.

Table I lists all the involved drivers (except for the graphic driver since it uses user-level driver) used in our experiments and the number of PCBs in each of them. We find that a driver typically has only around ten PCBs and each PCB has approximately 15 lines of code without making function calls (except the encryption and decryption functions). The total PCB code only account for $1 \backsim 3\%$ of the driver code. The tiny size of PCB and its simple logic allow for high security assurance, as compared to protecting the execution of thousands of lines of driver code.

9.4. Performance Evaluation

We experiment with DriverGaurd on a PC with Intel(R) Core(TM)2 Duo CPU E7200 @2.53GHz, 4GB main memory, running Xen 4.0.0 and a PV guest domain with Linux kernel 2.6.31.13. DriverGuard adds around 1.7K SLOC to the Xen hypervisor. Our performance evaluation includes a cost measurement of DriverGuard's component functions and a set of application tests with six devices. We remark that the I/O characteristic is favorable to our scheme as peripheral devices are usually much slower than the CPU. Therefore, DriverGuard does not affect the driver performance since the device speed is the performance bottleneck.

We choose 128-bit RC4 as the encryption cipher in our implementation rather than AES encryption, because RC4's compact code is easier to protect and does not significantly expand the PCB size.

9.4.1. Component Cost Evaluation. We instrument the DriverGuard code to measure the CPU cycles consumed by its main components including the escort hypercalls, the interrupt handler `do_IRQ`, the debug handler `do_debug`, the page fault handler `do_page_fault` and the general protection exception handler `do_general_protection` The results are shown in Table II. Note that the encryption cost within a PCB comprises the overhead of the secret key access that incurs one page fault and the hypervisor's checkpoint removal.

We also test the time cost induced by DriverGuard to data movements in the guest domain, including I/O port data and memory buffer transferring. The cost is due to the interceptions triggered by the checkpoints. We choose two commonly used functions: *inb* and *memcpy*. *inb* reads one byte from a serial port while we run *memcpy* to copy

Table II. Component Performance Results

Components	do_IRQ	do_debug	do_page_fault	do_general_protection	*Encryption 1KB*
CPU cycles	844	739	961	1813	23355

The extra cost of the DriverGuard components.

Table III. Data Access Results

	memcpy	*inb*
without DriverGuard (in CPU cycles)	1884	2738
with DriverGuard (in CPU cycles)	3026	2939
overhead (%)	1142 (60.62%)	201 (7.3%)

The time cost induced on the guest domain data access.

Table IV. Input Device Results

		W/O DriverGuard	**With DriverGuard**	**Overhead (%)**
Keyboard	key code transfer	0.053ms	0.138ms	0.085ms (160.40%)
Camera	Interrupt Handler	0.023ms	0.259ms	0.236ms (1026.09%)
	Waiting Time	33.24ms	33.38ms	0.14ms (0.42%)
Fingerprint-Reader	fingerprint collection	2.61s	2.63s	0.02s (0.77%)

The overhead of the protection on the keyboard, camera and fingerprint-reader I/O.

12K bytes from one buffer to another. Both the port and the memory buffer are protected by DriverGuard checkpoints and the functions are allowed to access. The test results are shown in Table III.

As shown in Table III, the overhead for protecting memory data flow is rather high (about 60%). Therefore, a severe performance drop will be seen in I/O flows involving frequent memory data movement. In fact, most device drivers are optimized to reduce the number of memory copying. A widely used practice is for the driver to maintain a large cache buffer and memory copying is invoked only when the cache is full.

9.4.2. Driver Performance Measurement. We test three input devices (keyboard, camera and fingerprint reader) and three output devices (printer, sound card, and the graphic card). For each device, we evaluate the performance overhead and latency for the device drivers and user applications.

Keyboard. When a user presses a key on the keyboard, an interrupt is generated by the hardware. The interrupt handler of the HID driver is invoked to get the key code and to send it into a tty buffer through the keyboard driver. Following that, an event is raised to trigger *sys_read*, which has been sleeping on the event. When being waked up, *sys_read* transfers the key value from the keyboard driver's buffer to a user space memory address. In our experiment, we measure the time cost of the interrupt handler that moves the data from the keyboard to the tty buffer. The results are shown in Table IV. Although the protected keyboard I/O is slower than the unprotected one, it does not affect the application because the overhead (i.e., 0.085ms) is still negligible as compared the speed of *human keystrokes*.

Camera. The web camera in our experiment is managed by the default Linux UVC driver. When the camera is opened by an application, it continuously collects video data and sends them to the application. The UVC driver's interrupt handler moves and decodes the captured data from the camera into a video frame, which resides in the driver's buffer mapped to the user space. The user application can directly use the frame data like normal user-space data without any kernel-to-user-space data movement.

Table V. Output Device Results

		Without DriverGuard	With DriverGuard	Overhead (%)
Printer	1 page	15.74s	16.19s	0.45s (2.86%)
	2 pages	27.36s	27.73s	0.37s (1.35%)
	4 pages	56.65s	58.15s	1.50s (2.65%)
	8 pages	120.75s	122.40s	1.65s (1.37%)
Sound Card	sound card open	7.8μs	12.3μs	4.5μs (57.7%)
Graphic Card	10subs	45.6 μs	45.8 μs	0.2 μs (0.44%)
	100subs	55.5 μs	55.5 μs	≈ 0

The overhead of the protection on the printer, sound card and graphic card I/O.

We run a command line program called *capture-example*[11] that reads the camera data continuously. We measure the time overhead of the UVC interrupt handler and the application's waiting time for getting new data, which is a key factor to the quality of the generated video stream. The results are shown in Table IV.

The interrupt handler's cost grows 10 times when under the protection of DriverGuard. The main overhead is due to the encryption on the camera data, which are 4 pages long. Nonetheless, the drivers spends much more time in waiting for the camera's data generation. Thus the cost of the interrupt handler does not cause the overall performance degradation. We test video chatting using Empathy 2.30.2, which is an graphic instant messenger. The experiment results do not show noticeable delays to the human users.

Fingerprint-Reader. Our fingerprint reader is the Upek Touchchip fingerprint sensor. In our evaluation experiment, we choose *Fingerprint GUI*[12] as the application which uses the default Linux driver *devio* to communicate with the fingerprint reader. When the fingerprint reader is active, the driver's interrupt handler continuously loads the collected fingerprint data into its buffers, which are then fetched by *Fingerprint GUI* by calling the *ioctl* function. In our experiments, we measure the whole I/O session of fingerprint collection. The results are shown in Table IV.

Printer. The printer in our experiments is HP Officejet 7210 and the device driver in use is *devio*. We use OpenOffice to print documents via a print-process running in the background. The print process opens the printer and issues *ioctl* to send data to the printer. After sending out the data, the print-process waits for a signal sent back by the printer to close the printer. In our experiments, we measure the turnaround time between the printer open and the printer close. The results are shown in Table V.

Sound Card. The sound card in our test is Intel Corporation 82801I (ICH9 Family) HD Audio and the driver in use is *HDA Intel*. We run the application *Totem* which plays MP3 files. Totem places its sound data into a user space buffer, which is mapped into the DMA buffer specified by the driver. When the music is in playing, Totem directly sends data into mapped DMA region in user space, and issues *ioctl* to synchronize and update information. The hardware fetches the data from the DMA buffer directly without the driver's involvement. Hence, DriverGuard is only involved in protecting the control region so that the kernel can not change the location of the DMA buffer in use.

Specifically, DriverGuard sets the sound card MMIO, the status and control region read-only after the probing stage. It rejects any update on the DMA descriptor base

[11]It can be downloaded from `http://v4l2spec.bytesex.org/spec/capture-example.html`.
[12]`http://www.n-view.net/Appliance/fingerprint/index.php`

address. DriverGuard also denies any access to the DMA buffer from the kernel. Therefore, there is no cost for DriverGuard during music playing, though the cost in opening the sound card is high, which is shown in Table V.

Graphic Card. We test DriverGuard with the graphic card. Since Xen 4.0.0 in our testing platform does not support Direct Rendering Manager (DRM), we have to run a guest Linux without DRM where the X Window sever directly manages all display outputs. The X Window server runs in the user space and does not use any kernel-level drivers. In a nutshell, it simply copies the display data to a designated memory buffer reserved by BIOS for the graphic card.

According to the design, we implement a loadable kernel module as the Trusted Loadable Module, which collects the reserved physical region, the user-space mapping region and the page table base address of the X Window server during the system booting. To defend the kernel's data stealing, DriverGuard grants the X Window server to access these protected regions and denies all accesses from the kernel or other user processes.

We test the display performance with *x11perf*, which is a graphic card performance measurement tool. We run the command *x11perf -repeat 100 -reps 10 -subs 10 100 -circulate* to measure the graphic card performance with and without DriverGuard protection. The results in Table V show the performance overhead is rather small. The reason is that when the X Window server accesses the protection region, there is only one page fault exception that is for the first access. After lifting the checkpoints, further access will not trigger any exception until it is switched off.

10. RELATED WORK

We first present and compare relevant trusted path schemes, and then we describe the user space protection approaches that are complementary with our work to protect the whole lifecycle of the I/O data. Finally, we describe the hypervisor security to illustrate the rationale of choosing the hypervisor as the root of trust.

10.1. Trusted Path

We attempt to categorize these relevant trusted path schemes according to the differences of their root of trust.

10.1.1. Virtualization-Based Trusted Path. The trusted path proposed by Zhou et al. [2012] aims to assure the *secrecy* and *authenticity* of I/O data transferred between a periphery device and an application. To build an exclusive trusted path between the device and the expected application, the hypervisor fixes the device configuration space which is achieved using Virtual Machine Control Structure/Block, Nested/Extended Page tables and IOMMU to prevent unauthorized software and DMA access, and interrupt delivery path which is achieved using Interrupt Remapping features and LAPIC x2APIC mode to make sure that the interrupt is delivered to expected handler. The expected applications are extended to support user-level drivers, which directly issue command to devices through the built trusted path. Obviously, it suffers compatibility issues since applications are numerous and any new application will need to be modified to satisfy the trusted path requirements. In comparison, DriverGuard requires modifications on driver code only. Moreover, DriverGuard does not need to defend against interrupt spoofing attack since it focuses on the *secrecy* of the I/O data between devices and applications, and only authorized PCB is able to access the decrypted I/O data. Even if the unauthorized codes are involved by unintended interrupt, they still can not access the protected I/O data.

The virtualization-based trusted path schemes are closely related to our work. BitVisor [Shinagawa et al. 2009] is a dedicated hypervisor to I/O management. It uses

a parapass-through mechanism whereby access operations on the monitored devices are intercepted and the operations on the other devices pass through without any checking. The interception allows the hypervisor to protect itself and to perform security functions on the device I/O. However, this approach does not protect the I/O data in the kernel space. BitVisor does not claim that they protect the MMIO mapping attacks.

Hypervisors with privileged root domains (e.g., the *Dom0* in Xen) are able to assign different device drivers to separate virtual machines (e.g., *driver domains* in Xen) and securely associate them with application virtual machines (e.g., *guest domains* in Xen) [Barham et al. 2003; Borders and Prakash 2007; Colp et al. 2011; Saroiu and Wolman 2010; Willmann et al. 2008]. These hypervisors isolate the device resources (i.e., I/O ports and the memory address-space, including MMIO regions) belonging to a device driver domain from other domains. Note that all these schemes do not consider the MMIO mapping attacks. Moreover, their TCBs enclose the whole operating system, which dramatically increases their trusted code bases.

10.1.2. Hardware-Based Trusted Path. The Zone Trusted Information Channel (ZTIC) [IBM Zurich Research Lab 2008] is a dedicated hardware, which provides a trusted path for users to confirm online transactions. The ZTIC-based trusted path completely bypasses the legacy channel in the users' computers. Bumpy [McCune et al. 2009] system proposes to protect user keyboard inputs by building a trust environment. It requires an encryption-capable keyboard and therefore is not applicable to generic devices.

The special-device-based schemes usually combine with cryptographic technology to protect the secrets in user and kernel space, such as Bumpy, which usually requires many mediations on the system. Such medications not only consiquently affect the compatibility of the scheme, but also often significantly reduce the usability, and even make the scheme impractical sometimes.

The UTP system [Filyanov et al. 2011] proposes an isolated kernel module to temporally manage user-centric I/O devices (e.g., keyboard and display) and enables a remote server to verify that a transaction summary is confirmed by a local keyboard input. BIND [Shi et al. 2005] binds data and code and uses cryptographic techniques to guarantee the integrity of data. However BIND is limited to derived data and cannot help on the confidentiality of the I/O data. Both of them require a hardware supported secure execution environment (e.g., secure kernel based on the AMD's Secure Execution Mode chip), which often occurs high latency and significant performance overhead.

10.1.3. OS-Based Trusted Path. Langweg [2004] propose a COTS-based scheme to solve the confidentiality, integrity, and authenticity of input and the output data.[13] Authors focus on the windows platform, where the Windows message mechanism is able be exploited by a malicious program to access the input and output messages (data) that are originally intended for other applications. By leveraging the advantages of the the DirextX, authors build a secure user interface to directly fetch input data and access the display hardware in exclusive mode. Trusted paths for browsers [Ye et al. 2005] focus on providing a trusted GUI to user, protecting user inputs to the intended browser. These two schemes only address security issues at the driver-applications interface, whereas the battlefield of DriverGuard is the entire I/O path. BitE [McCune et al. 2006] is an approach for preventing user-space malicious applications from accessing sensitive user input via a dedicated trusted path between input devices and the target application, and providing visual verification feedback to the user to prove that the input is really caught by the expected application.

[13]The confidentiality of the input data is not done.

All of them suffers from a large TCB since they are built atop large operating systems, and some of them even contains the Window Manager [McCune et al. 2006].

10.2. User-Space Protection

Terra [Garfinkel et al. 2003] protects the user applications by isolating them into separated secure domains, where the malicious applications and OSes are forbidden to access the secure domains. However TERRA systems occurs large computing base since it includes the whole secure domain into TCB. OverShadow [Chen et al. 2008] and SP^3 [Yang and Shin 2008] aim to protect the whole application execution against malicious application and OSes. However, both of them focus on the protection of the user space applications, instead of kernel space device drivers. Note that both of them can cooperate with DriverGuard to protect the whole life cycle of I/O data.

Flicker [McCune et al. 2008] system built on the TPM-based Dynamic Root Of Trust (DROT) technology can build an isolation environment to protect code and data. Due to the limitation of the TPM, the latency of the Flicker system is quite high. To minimize the latency, TrustVisor [McCune et al. 2010] scheme are proposed. By leveraging virtualization technology, TrustVisor virtualizes the physical TPM into Virtual TPM (VTPMs) and migrate them into hypervisor space. Note that both of them focus on the protection of a small piece of code and data. The increasing of the protection scope, such as protecting the whole application or device drivers, may lead both schemes to failure. In a word, both of them are not suitable for the I/O flow protection.

Lacuna [Dunn et al. 2012] aims to execute applications within *private session* and erase all execution traces once the session is *over* by leveraging the ephemeral channel. Specifically, the ephemeral channel connects the VM (applications inside) to hardware or software proxies to ensure that the sensitive data in the host OS is encrypted and only the endpoints can access the (plain text) data from private sessions.

10.3. Hypervisor Security

Comparing with legacy monolithic Operating Systems, the hypervisor is more secure since its size is relatively small and the exported attack surfaces for guest domains are considerably less. Although there have been several attacks discovered to compromise some versions of hypervisors [The Blue Pill; CVE-2008-0923 2008; King et al. 2006; Rafal et al. 2008], the security of the hypervisor can be enhanced through some existing mechanisms. The TPM-based authenticated boot can verify the integrity of the hypervisor when being launched, and the hardware-assisted virtualization technology, that is, Intel VT-x and AMD V, is able to significantly reduces the code size of the hypervisor, thereby the attack surface is reduced. Furthermore, there are some sophisticated framework systems [Azab et al. 2010; Rafal et al. 2008; Wang and Jiang 2010; Wang et al. 2010] proposed to enhance the security of the hypervisor. HyperGuard [Rafal et al. 2008], HyperCheck [Wang et al. 2010] and HyperSentry [Azab et al. 2010] are three System Management Mode (SMM)-based frameworks to measure and verify the integrity of hypervisors. The code for the SMM mode are protected by hardware chipset. HyperSafe [Wang and Jiang 2010] is a lightweight approach that protects existing bare-metal hypervisors with a unique self-protection capability to provide lifetime control flow integrity. In order to eliminate the programming bugs in the hypervisor, the rigorous formal verification mechanism [Heitmeyer et al. 2006] is able to be used to prove the correctness of the hypervisor.

11. CONCLUSION

We have proposed DriverGuard which is a hypervisor-based system protecting I/O flows between devices and applications, especially for devices generating data or rendering data. DriverGuard protects I/O device control, I/O data transfer and a driver's

data processing, against attacks from the untrusted guest kernel. It is featured with fine granularity protection, strong security assurance and low overhead. It only adds around 1.7K SLOC to the Xen hypervisor and a few lines to the driver code. DriverGuard can work jointly with user-space data protection schemes to safeguard the entire data lifecycle.

ACKNOWLEDGMENTS

We are grateful to Virgil Gligor and Adrian Perrig for their constructive suggestions during this work. We also thank anonymous reviewers for their valuable comments.

REFERENCES

Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C. 2010. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, New York, 38–49.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, 164–177.

Borders, K. and Prakash, A. 2007. Securing network input via a trusted input proxy. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Security (HOTSEC'07)*. USENIX Association, Berkeley, CA, 7:1–7:5.

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of CCS'08*. P. Syverson and S. Jha Eds., ACM, 27–38.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. 2010. Return-oriented programming without returns. In *Proceedings of CCS'10*. A. Keromytis and V. Shmatikov Eds., ACM, 559–72.

Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. K. 2008. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*.

Cheng, Y., Ding, X., and Deng, R. H. 2011. Driverguard: A fine-grained protection on I/O flows. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS'11)*. Springer-Verlag, Berlin, 227–244.

Chhabra, S., Rogers, B., Solihin, Y., and Prvulovic, M. 2011. Secureme: A hardware-software approach to full system security. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, 108–119.

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. 2001. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. ACM, New York, 73–88.

Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., and Warfield, A. 2011. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, 189–202.

CVE-2008-0923. 2008. http://cve.mitre.org/cgi-bin/cvename.cgi-?name=cve-2008-0923.

Dunn, A. M., Lee, M. Z., Jana, S., Kim, S., Silberstein, M., Xu, Y., Shmatikov, V., and Witchel, E. 2012. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 61–75.

Filyanov, A., McCune, J. M., Sadeghi, A.-R., and Winandy, M. 2011. Uni-directional trusted path: Transaction confirmation on just one device. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks*.

Fleming, S. 2008. Accessing PCI express configuration registers using intel chipsets. Tech. rep., Intel Corporation, http://www.intel.com/content/www/us/en/intelligent-systems/chipsets-pcie-config-reg-paper.html.

Ganapathy, V., Renzelmann, M. J., Balakrishnan, A., Swift, M. M., and Jha, S. 2008. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, 168–178.

Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. 2003. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*. ACM, New York, 93–206.

Heintze, N. and Tardieu, O. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, New York, 254–263.

Heitmeyer, C. L., Archer, M., Leonard, E. I., and McLean, J. 2006. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*. ACM, New York, 346–355.

IBM Zurich Research Lab. 2008. Security on a stick.

Intel. 2008. Intel I/O controller hub 9 (ICH9) family datasheet.

Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. 2012. LIBDFT: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*. ACM, New York, 121–132.

King, S. T., Chen, P. M., Wang, Y.-M., Verbowski, C., Wang, H. J., and Lorch, J. R. 2006. Subvirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, 314–327.

Kun, S., Jiang, W., Fengwei, Z., and Angelos, S. 2012. SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*.

Langweg, H. 2004. Building a trusted path for applications using cots components. In *Proceedings of NATO RTO IST Panel Symposium on Adaptive Defence in Unclassified Networks*.

Li, Y., McCune, J. M., and Perrig, A. 2011. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, New York, 3–16.

Lineberry, A. 2009. Malicious code injection via /dev/mem. In *Black Hat*.

McCune, J. M., Perrig, A., and Reiter, M. K. 2006. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the Annual Conference on USENIX'06 Annual Technical Conference*. USENIX Association, Berkeley, CA, 17–17.

McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. 2008. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*.

McCune, J. M., Perrig, A., and Reiter, M. K. 2009. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*.

McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. 2010. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*. IEEE Computer Society, Los Alamitos, CA, 143–158.

Mock, M., Atkinson, D. C., Chambers, C., and Eggers, S. J. 2002. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes 27*, 6, 71–80.

Newsome, J. and Song, D. 2005. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*.

Payne, B. D., Carbone, M., Sharif, M., and Lee, W. 2008. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, 233–247.

Phoenix Technologies. 2006. TrustedCore: Foundation for secure CRTM and BIOS implementation. https://forms.phoenix.com/whitepaperdownload-/docs/trustedcore_wp.pdf.

Rafal, W., Joanna, R., and Alexander, T. 2008. Xen owning trilogy. website. http://invisible-thingslab.com/itl/Resources.html.

Santelices, R., Zhang, Y., Jiang, S., Cai, H., and jie Zhang, Y. 2012. Quantitative program slicing: Separating statements by relevance. Tech. rep.

Saroiu, S. and Wolman, A. 2010. I am a sensor, and I approve this message. In *Proceedings of the 11th Workshop on Mobile Computing Systems & Applications (HotMobile'10)*. ACM, New York, 37–42.

Seshadri, A., Luk, M., Qu, N., and Perrig, A. 2007. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, 335–350.

Shacham, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS'07*. S. De Capitani di Vimercati and P. Syverson Eds., ACM, 552–61.

Shi, E., Perrig, A., and Doorn, L. V. 2005. BIND: A fine-grained attestation service for secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. 154–168.

Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., and Kato, K. 2009. Bitvisor: A thin hypervisor for enforcing I/O device security. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*. ACM, New York, 121–130.

Song, D. X., Wagner, D., and Tian, X. 2001. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Berkeley, CA, 25–25.

Sridharan, M., Fink, S. J., and Bodik, R. 2007. Thin slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, 112–122.

Swift, M. M., Bershad, B. N., and Levy, H. M. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, 207–222.

The Blue Pill. http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

Vasudevan, A., Parno, B., Qu, N., Gligor, V. D., and Perrig, A. 2012. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST'12)*. Springer-Verlag, Berlin, 34–54.

Wang, J., Stavrou, A., and Ghosh, A. 2010. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection (RAID'10)*. Springer-Verlag, Berlin, 158–177.

Wang, Z. and Jiang, X. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*. IEEE Computer Society, Los Alamitos, CA, 380–395.

Weiser, M. D. 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, AAI8007856, Ann Arbor, MI.

White, A. M., Matthews, A. R., Snow, K. Z., and Monrose, F. 2011. Phonotactic reconstruction of encrypted VOIP conversations: Hookt on fon-iks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'11)*. IEEE Computer Society, Los Alamitos, CA, 3–18.

Willmann, P., Rixner, S., and Cox, A. L. 2008. Protection strategies for direct access to virtualized I/O devices. In *Proceedings of the USENIX Annual Technical Conference*.

Yang, J. and Shin, K. G. 2008. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'08)*. ACM, New York, 71–80.

Ye, Z. E., Smith, S., and Anthony, D. 2005. Trusted paths for browsers. *ACM Trans. Inf. Syst. Secur. 8*, 2, 153–186.

Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., and Brewer, E. 2006. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 45–60.

Zhou, Z., Gligor, V. D., Newsome, J., and McCune, J. M. 2012. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Received June 2012; revised January 2013, May 2013; accepted June 2013